SMART CONTRACT AUDIT REPORT

for

Torah Protocol

Prepared By: Xiaomi Huang

PeckShield

November 25, 2022

Document Properties

Client	Torah
Title	Smart Contract Audit Report
Target	Torah
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 25, 2022	Xuxian Jiang	Final Release
1.0-rc2	November 21, 2022	Xuxian Jiang	Release Candidate #2
1.0-rc1	November 15, 2022	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Torah` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Torah

`Torah Finance` is a decentralized exchange of liquidity pools on Ethereum. It is designed to reward users so that the number of `TRH` rewarded for trading will vary with the total trading volume. `Torah` allows the liquidity returns to be obtained through the funds deposited into the flow pool. `VeTRH` is the equity token of `Torah Finance`, with both accelerating and voting interests. `Torah`'s incentive is the ability to increase your rewards for liquidity offered or trading rewards. This audit only focuses on the swap contracts that are derived from the popular `Curve` protocol and the reward distribution is not part of the audit. The basic information of `Torah Finance` is as follows:

Table 1.1: Basic Information of Torah

Item	Description
Target	Torah
Website	https://torah.finance/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 25, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/torah-fi/exchange.git (053299d)
- https://github.com/torah-fi/contracts.git (fd88213)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/torah-fi/exchange.git (280953f)
- https://github.com/torah-fi/contracts.git (fd88213)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Torah Finance` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	1	■
Informational	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Torah Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Logic in Crypto3PoolView::get_dy()/calc_token_amount()	Business Logic	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Informational	Suggested EIP2612 Support in Plain2/3/4/Meta/Balances.vy	Coding Practices	Confirmed
PVE-004	Medium	Improper Initialization of CurveTokenV5	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic in Crypto3PoolView::get_dy()/calc_token_amount()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Crypto3PoolView
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Torah Finance protocol is forked from Curve for the token exchange support. While reviewing the forked version, we notice it is based on an earlier version and we strongly suggest to upgrade to the latest one.

For elaboration, we show below the related Crypto3PoolView.vy contract, which is based on the version 0.2.12 of the curve-crypto-contract repository, which now has the latest version of 0.3.1. The latest version includes a variety of improvements, including the dynamic D calculation in core get_dy() and calc_token_amount() routines.

```
@external
@view
def get_dy(i: uint256, j: uint256, dx: uint256) -> uint256:
    assert i != j and i < N_COINS and j < N_COINS, "coin index out of range"
    assert dx > 0, "do not exchange 0 coins"

    precisions: uint256[N_COINS] = PRECISIONS

    price_scale: uint256[N_COINS-1] = empty(uint256[N_COINS-1])
    for k in range(N_COINS-1):
        price_scale[k] = Curve(msg.sender).price_scale(k)
    xp: uint256[N_COINS] = empty(uint256[N_COINS])
    for k in range(N_COINS):
        xp[k] = Curve(msg.sender).balances(k)
```

```
    xp[i] += dx
    xp[0] *= precisions[0]
    for k in range(N_COINS-1):
        xp[k+1] = xp[k+1] * price_scale[k] * precisions[k+1] / PRECISION

    A: uint256 = Curve(msg.sender).A()
    gamma: uint256 = Curve(msg.sender).gamma()

    y: uint256 = Math(self.math).newton_y(A, gamma, xp, Curve(msg.sender).D(), j)
    dy: uint256 = xp[j] - y - 1
    xp[j] = y
    if j > 0:
        dy = dy * PRECISION / price_scale[j-1]
    dy /= precisions[j]
    dy -= Curve(msg.sender).fee_calc(xp) * dy / 10**10

    return dy
```

Listing 3.1: `Crypto3PoolView::get_dy()`

Recommendation Update the base implementation to the latest version.

Status This issue has been fixed in this commit: `cd4d3d9`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event.*

The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `add_liquidity()` routine in the `DepositZap.vy` contract. If the `USDT` token is supported as one of the base token, the unsafe version of `ERC20(coin).transferFrom(msg.sender, self, _deposit_amounts[0])` (lines 92 and 103) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
67 @external
68 def add_liquidity(
69         _pool: address,
70         _deposit_amounts: uint256[N_ALL_COINS],
71         _min_mint_amount: uint256,
72         _receiver: address = msg.sender,
73 ) -> uint256:
74     """
75     @notice Wrap underlying coins and deposit them into `_pool`
76     @param _pool Address of the pool to deposit into
77     @param _deposit_amounts List of amounts of underlying coins to deposit
78     @param _min_mint_amount Minimum amount of LP tokens to mint from the deposit
```

```
    @param _receiver Address that receives the LP tokens
    @return Amount of LP tokens received by depositing
    """
    meta_amounts: uint256[N_COINS] = empty(uint256[N_COINS])
    base_amounts: uint256[BASE_N_COINS] = empty(uint256[BASE_N_COINS])
    deposit_base: bool = False
    base_coins: address[3] = self.base_coins

    if _deposit_amounts[0] != 0:
        coin: address = CurveMeta(_pool).coins(0)
        if not self.is_approved[coin][_pool]:
            ERC20(coin).approve(_pool, MAX_UINT256)
            self.is_approved[coin][_pool] = True
        ERC20(coin).transferFrom(msg.sender, self, _deposit_amounts[0])
        meta_amounts[0] = _deposit_amounts[0]

    for i in range(1, N_ALL_COINS):
        amount: uint256 = _deposit_amounts[i]
        if amount == 0:
            continue
        deposit_base = True
        base_idx: uint256 = i - 1
        coin: address = base_coins[base_idx]

        ERC20(coin).transferFrom(msg.sender, self, amount)
        # Handle potential Tether fees
        if i == N_ALL_COINS - 1:
            base_amounts[base_idx] = ERC20(coin).balanceOf(self)
        else:
            base_amounts[base_idx] = amount

    # Deposit to the base pool
    if deposit_base:
        coin: address = self.base_lp_token
        CurveBase(self.base_pool).add_liquidity(base_amounts, 0)
        meta_amounts[MAX_COIN] = ERC20(coin).balanceOf(self)
        if not self.is_approved[coin][_pool]:
            ERC20(coin).approve(_pool, MAX_UINT256)
            self.is_approved[coin][_pool] = True

    # Deposit to the meta pool
    return CurveMeta(_pool).add_liquidity(meta_amounts, _min_mint_amount, _receiver)
```

Listing 3.3: `DepositZap::add_liquidity()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in this commit: `cd4d3d9`.

3.3 Suggested EIP2612 Support in Plain2/3/4/Meta/Balances.vy

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The various pools in `Torah` build efficient swap curves that are designed to work with various ERC20-compliant tokens. While the interplay among the standard ERC20 APIs, i.e., `approve()` and `transferFrom()`, allows for tokens to not only be transferred between externally owned accounts (EOA), but to be used in other contracts under application specific conditions by abstracting away `msg.sender` as the defining mechanism for token access control.

However, a potential improvement is the adoption of `EIP2612`, which extends the EIP20 standard with a new function `permit()` so that users are allowed to modify the allowance mapping using a signed message, instead of through `msg.sender`. In the following, we use the `Plain2Balances.vy` contract as an example. This contract is designed to swap two stablecoins without lending. The addition of `permit()` can greatly improve user experience,

```
@external
def permit(
    _owner: address,
    _spender: address,
    _value: uint256,
    _deadline: uint256,
    _v: uint8,
    _r: bytes32,
    _s: bytes32
) -> bool:
    """
    @notice Approves spender by owner's signature to expend owner's tokens.
        See https://eips.ethereum.org/EIPS/eip-2612.
    @dev Inspired by https://github.com/yearn/yearn-vaults/blob/main/contracts/Vault.vy#
        L753-L793
    @dev Supports smart contract wallets which implement ERC1271
        https://eips.ethereum.org/EIPS/eip-1271
    @param _owner The address which is a source of funds and has signed the Permit.
    @param _spender The address which is allowed to spend the funds.
    @param _value The amount of tokens to be spent.
    @param _deadline The timestamp after which the Permit is no longer valid.
    @param _v The bytes[64] of the valid secp256k1 signature of permit by owner
    @param _r The bytes[0:32] of the valid secp256k1 signature of permit by owner
```

```
    @param _s The bytes[32:64] of the valid secp256k1 signature of permit by owner
    @return True, if transaction completes successfully
    """
    assert _owner != ZERO_ADDRESS
    assert block.timestamp <= _deadline

    nonce: uint256 = self.nonces[_owner]
    digest: bytes32 = keccak256(
        concat(
            b"\x19\x01",
            self.DOMAIN_SEPARATOR,
            keccak256(_abi_encode(PERMIT_TYPEHASH, _owner, _spender, _value, nonce, _deadline))
        )
    )

    if _owner.is_contract:
        sig: Bytes[65] = concat(_abi_encode(_r, _s), slice(convert(_v, bytes32), 31, 1))
        # reentrancy not a concern since this is a staticcall
        assert ERC1271(_owner).isValidSignature(digest, sig) == ERC1271_MAGIC_VAL
    else:
        assert ecrecover(digest, convert(_v, uint256), convert(_r, uint256), convert(_s, uint256)) == _owner

    self.allowance[_owner][_spender] = _value
    self.nonces[_owner] = nonce + 1

    log Approval(_owner, _spender, _value)
    return True
```

Listing 3.4: `Plain2Balances::permit()`

Recommendation Support the `EIP2612` specification with the `permit()` implementation. The same suggestion is also applicable to other pool contracts, including `Plain3Balances.vy`, `Plain4Balances.vy`, and `MetaUSDBalances.vy`.

Status This issue has been confirmed.

3.4 Improper Initialization of CurveTokenV5

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CurveTokenV5`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Torah Finance` protocol has a `CurveTokenV5` contract, which is designed to act as the pool LP token contract. Our analysis shows that this token contract unexpectedly initializes its `minter` state to `0x0000000000000000000000000000000000000001`, which effectively blocks the built-in functions, including `mint()`, `mint_relative()`, `burnFrom()`, and `initialize()`.

To elaborate, we show below the full implementation of the `__init__()`. Note it is a reasonable assumption that no one knows the private key behind the `0x0000000000000000000000000000000000000001` address. As a result, it effectively disables the current privileged functions that can only be exercised by the admin account!

```
@external
def __init__():
    self.minter = 0x0000000000000000000000000000000000000001
```

Listing 3.5: `CurveTokenV5::__init__()`

```
@external
def mint(_to: address, _value: uint256) -> bool:
    """
    @dev Mint an amount of the token and assigns it to an account.
         This encapsulates the modification of balances such that the
         proper events are emitted.
    @param _to The account that will receive the created tokens.
    @param _value The amount that will be created.
    """
    assert msg.sender == self.minter

    self.totalSupply += _value
    self.balanceOf[_to] += _value

    log Transfer(ZERO_ADDRESS, _to, _value)
    return True


@external
def mint_relative(_to: address, frac: uint256) -> uint256:
    """
    @dev Increases supply by factor of (1 + frac/1e18) and mints it for _to
```

```
    """
    assert msg.sender == self.minter

    supply: uint256 = self.totalSupply
    d_supply: uint256 = supply * frac / 10**18
    if d_supply > 0:
        self.totalSupply = supply + d_supply
        self.balanceOf[_to] += d_supply
        log Transfer(ZERO_ADDRESS, _to, d_supply)

    return d_supply


@external
def burnFrom(_to: address, _value: uint256) -> bool:
    """
    @dev Burn an amount of the token from a given account.
    @param _to The account whose tokens will be burned.
    @param _value The amount that will be burned.
    """
    assert msg.sender == self.minter

    self.totalSupply -= _value
    self.balanceOf[_to] -= _value

    log Transfer(_to, ZERO_ADDRESS, _value)
    return True
```

Listing 3.6: Privileged Functions in `CurveTokenV5`

Recommendation Revisit the minter management in the above `CurveTokenV5` contract.

Status This issue has been resolved as the deployment is made via `create_forwarder_to()` and initialized via the `initialize()`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Torah Finance` protocol, there is a privileged account, i.e., `admin`, which plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and reward adjustment).

It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
@external
def add_pool(
        _pool: address,
        _n_coins: uint256,
        _lp_token: address,
        _rate_info: bytes32,
        _decimals: uint256,
        _underlying_decimals: uint256,
        _has_initial_A: bool,
        _is_v1: bool,
        _name: String[64],
):
    """
    @notice Add a pool to the registry
    @dev Only callable by admin
    @param _pool Pool address to add
    @param _n_coins Number of coins in the pool
    @param _lp_token Pool deposit token address
    @param _rate_info Encoded twenty-byte rate calculator address and/or four-byte
        function signature to query coin rates
    @param _decimals Coin decimal values, tightly packed as uint8 in a little-endian
        bytes32
    @param _underlying_decimals Underlying coin decimal values, tightly packed
                                as uint8 in a little-endian bytes32
    @param _name The name of the pool
    """
    self._add_pool(
        msg.sender,
        _pool,
        _n_coins + shift(_n_coins, 128),
        _lp_token,
        _rate_info,
        _has_initial_A,
        _is_v1,
        _name,
        )

    coins: address[MAX_COINS] = self._get_new_pool_coins(_pool, _n_coins, False, _is_v1)
    decimals: uint256 = _decimals
    if decimals == 0:
        decimals = self._get_new_pool_decimals(coins, _n_coins)
    self.pool_data[_pool].decimals = decimals

    coins = self._get_new_pool_coins(_pool, _n_coins, True, _is_v1)
    decimals = _underlying_decimals
    if decimals == 0:
        decimals = self._get_new_pool_decimals(coins, _n_coins)
    self.pool_data[_pool].underlying_decimals = decimals
```

```


@external
def add_pool_without_underlying(
        _pool: address,
        _n_coins: uint256,
        _lp_token: address,
        _rate_info: bytes32,
        _decimals: uint256,
        _use_rates: uint256,
        _has_initial_A: bool,
        _is_v1: bool,
        _name: String[64],
):
    ...
}
```

Listing 3.7: Example `Setters` in the `PoolRegistry`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been mitigated. The team decides to use multi-sig contract for the privileged `admin` account.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Torah Finance` protocol, which is a decentralized exchange of liquidity pools on Ethereum. It is designed to reward users so that the number of `TRH` rewarded for trading will vary with the total trading volume. `Torah` allows the liquidity returns to be obtained through the funds deposited into the flow pool. `VeTRH` is the equity token of `Torah Finance`, with both accelerating and voting interests. `Torah`'s incentive is the ability to increase your rewards for liquidity offered or trading rewards. This audit only focuses on the swap contracts that are derived from the popular `Curve` protocol. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.